

# Strong Typed Böhm Theorem and Functional Completeness on the Linear Lambda Calculus

Satoshi Matsuoka

National Institute of Advanced Industrial Science and Technology (AIST),  
1-1-1 Umezono, Tsukuba, Ibaraki, 305-8565 Japan  
matsuoka@ni.aist.go.jp

In this paper, we prove a version of the typed Böhm theorem on the linear lambda calculus, which says, for any given types  $A$  and  $B$ , when two different closed terms  $s_1$  and  $s_2$  of  $A$  and any closed terms  $u_1$  and  $u_2$  of  $B$  are given, there is a term  $t$  such that  $t s_1$  is convertible to  $u_1$  and  $t s_2$  is convertible to  $u_2$ . Several years ago, a weaker version of this theorem was proved, but the stronger version was open. As a corollary of this theorem, we prove that if  $A$  has two different closed terms  $s_1$  and  $s_2$ , then  $A$  is functionally complete with regard to  $s_1$  and  $s_2$ . So far, it was only known that a few types are functionally complete.

## 1 Introduction

This paper is an addendum to the paper [13], which was published several years ago. The previous paper establishes the following result in the linear  $\lambda$ -calculus:

For any type  $A$  and two different closed terms  $s_1$  and  $s_2$  of type  $A$ , there is a term  $t$  such that

$$t s_1 =_{\beta\eta c} \underline{0} \quad \text{and} \quad t s_2 =_{\beta\eta c} \underline{1},$$

where  $\underline{0} \equiv_{\text{def}} \lambda x. \lambda f. \lambda g. f(g(x))$  and  $\underline{1} \equiv_{\text{def}} \lambda x. \lambda f. \lambda g. g(f(x))$ .

In [13], the proof net notation for the intuitionistic multiplicative linear logic (for short, IMLL) was used, but as shown later, the linear  $\lambda$ -calculus can be regarded as a subsystem of IMLL proof nets. In addition the equality  $=_{\beta\eta c}$  will be defined precisely later. In this paper, we prove a stronger version of the previous statement, which is stated as follows:

For any given types  $A$  and  $B$ , when two different closed terms  $s_1$  and  $s_2$  of  $A$  and any closed terms  $u_1$  and  $u_2$  of  $B$  are given, there is a term  $t$  such that

$$t s_1 =_{\beta\eta c} u_1 \quad \text{and} \quad t s_2 =_{\beta\eta c} u_2.$$

The stronger version was an open question in [13]. Note that the strong version is trivially derived from the weak one in the simply typed  $\lambda$ -calculus, because the calculus allows discard and copy of variables freely. But the linear  $\lambda$ -calculus officially does not allow these two operations. So some technical devices are required. The basic idea of our solution is to extend the typability by a linear implicational formula  $A \multimap B$  to a more liberalized form. We call the extended typability *poly-typability*, which is a mathematical formulation of the typing discipline used in [12]. Thanks to the extension, we can prove Projection Lemma (Lemma 5.1) and Constant Function Lemma (Lemma 5.2), which are the keys to establish our typed Böhm theorem.

One application is the functional completeness problem of the linear  $\lambda$ -calculus. It raises the question about the possibility of Boolean representability in the linear  $\lambda$ -calculus. We prove that any type with at least two different closed terms is functionally complete. This means that any two-valued functions can be represented over these two terms. So far, it was only known that a few types have this property. Our functional completeness theorem liberalizes us from sticking to specific types. This situation is analogous to that of the degree of freedom about a base choice in linear algebra: linear independence is enough. Similarly we may choose any different two terms of any type in order to establish the functional completeness.

The strong typed Böhm theorem gives a general construction of linear  $\lambda$ -terms that satisfy a given specification for inputs and outputs. It is expected that useful theorems about linear  $\lambda$ -terms will be proved by using the theorem further.

**Comparison with the case of the simply typed lambda calculus** The first proof of the typed Böhm theorem for the simply typed lambda calculus was given in [17]. The proof is based on the *reducibility theorem* in [16] (see also Theorem 3.4.8 in [1]). Our proof proceeds in a similar manner to Statman's proof. But the proof of the reducibility theorem is rather complicated, since it uses different operations. On the other hand, the proof of our analogue, which is Proposition 3.1, is much simpler, because our proof is based on one simple principle, i.e., linear distributive law (see, e.g., [3]) <sup>1</sup>:

$$((A \wp B) \otimes C) \multimap (A \wp (B \otimes C))$$

On the other hand, while the final separation argument of Statman's proof only uses type instantiation, our proof of Theorem 5.1 needs the notion of poly-types.

## 2 Typing Rules, Reduction Rules, and an Equational Theory

In this section we give our type assignment system for the linear  $\lambda$ -calculus and discuss some reduction rules and equivalence relations on the typed terms of the system. Our system is based on the natural deduction calculus given in [19], which is equivalent to the system based on the sequent calculus or proof nets in [6] (e.g., see [19]). Our notation is the same as that in [12]: the reader can confirm our results using an implementation of Standard ML [15].

### Types

$$A ::= 'a \mid A1 * A2 \mid A1 \multimap A2$$

The symbol  $'a$  stands for a type variable. On the other hand  $A1 * A2$  stands for the tensor product  $A1 \otimes A2$  and  $A1 \multimap A2$  for the linear implication  $A1 \multimap A2$  in the usual notation.

**Terms** We use  $x, y, z$  for term variables and  $r, s, t, u, v, w$  for general terms.

**Linear Typing Contexts** A linear typing context is a finite list of pairs  $x:A$  such that each variable occurs in the list once. Usually we use Greek letters  $\Gamma, \Delta, \dots$  to denote linear typing contexts.

---

<sup>1</sup>For example, this principle includes  $((A \multimap B) \multimap C) \multimap D \multimap (A \multimap (B \multimap C) \multimap D)$ ,  $((A \multimap B) \otimes C) \multimap (A \multimap (B \otimes C))$ , and  $((A \multimap B \otimes C) \multimap D) \multimap (B \multimap (A \multimap C) \multimap D)$ . This observation was the starting point of Proposition 3.1.

**Type Assignment System**

$$\begin{array}{c}
\frac{}{x:A \vdash x:A} \quad \frac{\Gamma, x:A, y:B, \Delta \vdash t:C}{\Gamma, y:B, x:A, \Delta \vdash t:C} \\
\frac{x:A, \Gamma \vdash t:B}{\Gamma \vdash \text{fn } x \Rightarrow t : A \rightarrow B} \quad \frac{\Gamma \vdash t : A \rightarrow B \quad \Delta \vdash s : A}{\Gamma, \Delta \vdash ts : B} \\
\frac{\Gamma \vdash s : A \quad \Delta \vdash t : B}{\Gamma, \Delta \vdash (s, t) : A * B} \quad \frac{\Gamma \vdash s : A * B \quad x:A, y:B, \Delta \vdash t : C}{\Gamma, \Delta \vdash \text{let val } (x, y) = s \text{ in } t \text{ end} : C}
\end{array}$$

In addition we assume that for each term variable, if an occurrence of the variable appears in a sequent in a term derivation, then the number of the occurrences in the sequent is exactly two. For a term  $t$  the set of bound variables  $BV(t)$  is defined recursively as follows:

- $BV(x) = \emptyset$ ,
- $BV(st) = BV((s, t)) = BV(t) \cup BV(s)$ ,
- $BV(\text{fn } x \Rightarrow t) = \{x\} \cup BV(t)$ ,
- $BV(\text{let val } (x, y) = s \text{ in } t \text{ end}) = \{x, y\} \cup BV(s) \cup BV(t)$ .

The set of free variables of  $t$ , denoted by  $FV(t)$  is the complement of the set of variables in  $t$  with respect to  $BV(t)$ . The function declaration

`fun f x1 x2 ... xn = t`

is interpreted as the following term:

`f = fn x1 => fn x2 => ... => fn xn => t`

Below we consider only closed terms (i.e. combinators)

$\vdash t : A$ .

**Term Reduction Rules** Two of our reduction rules are

$(\beta_1): (\text{fn } x \Rightarrow t)s \Rightarrow_{\beta_1} t[s/x]$

$(\beta_2): \text{let val } (x, y) = (u, v) \text{ in } w \text{ end} \Rightarrow_{\beta_2} w[u/x, v/y]$

Then note that if a function  $f$  is defined by

`fun f x1 x2 ... xn = t`

and

$x_1:A_1, \dots, x_n:A_n \mid -t:B, \quad \mid -t_1:A_1, \dots, \mid -t_n:A_n$

then, we have

$f \ t_1 \ \dots \ t_n \Rightarrow_{\beta_1}^* t[t_1/x_1, \dots, t_n/x_n]$ .

We denote the reflexive transitive closure of a relation  $R$  by  $R^*$ . In the following  $\rightarrow_\beta$  denotes the congruent (one-step reduction) relation generated by the two reduction rules above and the following contexts:

$$\begin{aligned}
C[] &= [] \mid C[]t \mid tC[] \mid (t, C[]) \mid (C[], t) \mid \text{fn } x \Rightarrow C[] \\
&\quad \mid \text{let val } (x, y) = C[] \text{ in } t \text{ end} \mid \text{let val } (x, y) = t \text{ in } C[] \text{ end}
\end{aligned}$$

We define the set of variables captured by a context  $C[]$ , denoted by  $CV(C[])$  recursively:

- $CV([]) = \emptyset$ ,
- $CV(C[]t) = CV(tC[]) = CV((t, C[])) = CV((C[], t)) = CV(C[])$ ,
- $CV(\text{fn } x \Rightarrow C[]) = \{x\} \cup CV(C[])$ ,
- $CV(\text{let val } (x, y) = C[] \text{ in } t \text{ end}) = CV(C[])$ ,

- $CV(\text{let val } (x,y) = t \text{ in } C[] \text{ end}) = \{x,y\} \cup CV(C[])$ .

The set of free variables of a context  $C[]$ , denoted by  $FV(C[])$  is defined similarly to that of a term  $t$ .

In order to establish a full and faithful embedding from linear  $\lambda$ -terms into IMLL proof nets, we introduce further reduction rules. Basically we follow [11], but note that a simpler presentation is given than that of [11], following a suggestion of an anonymous referee. The following are  $\eta$ -rules:

$$(\eta_1): \text{fn } x \Rightarrow (t \ x) \Rightarrow_{\eta_1} t$$

$$(\eta_2): \text{let val } (x,y) = t \text{ in } (x,y) \Rightarrow_{\eta_2} t$$

In the following  $\rightarrow_{\beta\eta}$  denotes the congruent (one-step reduction) relation generated by the four reduction rules above and any context  $C[]$ . But these reduction rules are not enough: different normal terms may correspond to the same normal IMLL proof net. In order to make further identification we introduce the following commutative conversion rule. Then we define the commutative conversion relation  $\leftrightarrow_c$ :

$$C[\text{let val } (x,y)=t \text{ in } u \text{ end}] \leftrightarrow_c \text{let val } (x,y)=t \text{ in } C[u] \text{ end}$$

where  $FV(C[]) \cap \{x,y\} = \emptyset$  and  $CV(C[]) \cap FV(t) = \emptyset$

Let  $=_c$  be the congruent equivalence relation generated by  $\leftrightarrow_c$  and any context  $C[]$ . Then we define  $\rightarrow_{\beta\eta c}$  as the least relation satisfying the following rule:

$$\frac{t =_c t' \quad t' \rightarrow_{\beta\eta} u' \quad u' =_c u}{t \rightarrow_{\beta\eta c} u}$$

Then the following holds.

**Proposition 2.1 (Church Rosser[11])** *if  $t \rightarrow_{\beta\eta c} t'$  and  $t \rightarrow_{\beta\eta c} u'$  then for some  $w =_c w'$ ,  $t' \rightarrow_{\beta\eta c} w$  and  $u' \rightarrow_{\beta\eta c} w'$ .*

Furthermore we can easily prove that  $\rightarrow_{\beta\eta c}$  is strong normalizable as shown in [11]. We can conclude that we have the uniqueness property for normal forms under  $\rightarrow_{\beta\eta c}$  up to  $=_c$ .

**Equality Rules** Next we define our fundamental equality  $=_{\beta\eta c}$ , which is given in [11] implicitly. The equality  $=_{\beta\eta c}$  is the smallest relation satisfying the following rules of the three groups:

(Relation Group)

$$(\text{Ref}) \frac{\Gamma \vdash t : A}{\Gamma \vdash t = t : A} \quad (\text{Sym}) \frac{\Gamma \vdash t = s : A}{\Gamma \vdash s = t : A} \quad (\text{Trans}) \frac{\Gamma \vdash t = s : A \quad \Gamma \vdash s = u : A}{\Gamma \vdash t = u : A}$$

(Reduction Group)

$$(\text{Eqc}) \frac{\Gamma \vdash t : A \quad t \leftrightarrow_c t'}{\Gamma \vdash t = t' : A} \quad (\text{Eq}\beta\eta) \frac{\Gamma \vdash t : A \quad t \rightarrow_{\beta\eta c} t'}{\Gamma \vdash t = t' : A}$$

(Congruence Group)

$$(\text{Eq}\lambda) \frac{x : A, \Gamma \vdash t = t' : B}{\Gamma \vdash \text{fn } x \Rightarrow t = \text{fn } x \Rightarrow t' : A \rightarrow B}$$

$$(\text{Eqap}) \frac{\Gamma \vdash t = t' : A \rightarrow B \quad \Delta \vdash s = s' : A}{\Gamma, \Delta \vdash ts = t's' : B}$$

$$(\text{Eq tup}) \frac{\Gamma \vdash s = s' : A \quad \Delta \vdash t = t' : B}{\Gamma, \Delta \vdash (s, t) = (s', t') : A * B}$$

$$(\text{Eqlet}) \frac{\Gamma \vdash s = s' : A * B \quad x : A, y : B, \Delta \vdash t = t' : C}{\Gamma, \Delta \vdash \text{let val } (x,y)=s \text{ in } t \text{ end} = \text{let val } (x,y)=s' \text{ in } t' \text{ end} : C}$$

**The relationship between linear  $\lambda$  terms and IMLL proof nets** We can prove the existence of a full and faithful embedding from the equivalence classes of linear  $\lambda$ -terms up to  $=_{\beta\eta c}$  into the set of normal IMLL proof nets in the sense of [13]. The proof is given in Appendix A.1 with a brief introduction to IMLL proof nets.

### 3 The Linear Distributive Transformation

In this section we recall some definitions and results in [13]. In [13], most results are given by IMLL proof nets, not by the linear  $\lambda$ -calculus. But we have already given a full and faithful embedding from linear  $\lambda$ -terms to IMLL proof nets. So those results can be used for the linear  $\lambda$ -calculus freely.

**Definition 3.1** A linear  $\lambda$ -term  $t$  is *implicational* if there are neither `let` constructors nor  $(-, -)$  constructors in  $t$ .

A type  $A$  is *implicational* if there are no  $A_1 * A_2$  tensor subformulas in  $A$ . The order of an implicational formula  $A$ ,  $\text{order}(A)$  is defined inductively as follows:

1.  $A$  is a propositional variable  $'a$ , then  $\text{order}(A) = 1$ .
2.  $A$  is  $A_1 \rightarrow \dots \rightarrow A_n \rightarrow 'a$ , then  $\text{order}(A)$  is

$$\max\{\text{order}(A_1), \dots, \text{order}(A_n)\} + 1$$

The following proposition is the linear lambda calculus version of Corollary 2 in [13], which says that any different two terms of a type can be mapped into different two terms of another (but possibly the same) type with lower order (more precisely, less than 4) without any tensor connectives injectively. The purpose is to transform given terms into terms that can be treated easily.

**Proposition 3.1 (Linear Distributive Transformation)** Let  $A$  be a type and  $s_1$  and  $s_2$  be two different closed terms of  $A$  up to  $=_{\beta\eta c}$ . Then there is a linear  $\lambda$ -term  $\text{LDTr}_A$  such that  $\text{LDTr}_A s_1 \neq_{\beta\eta c} \text{LDTr}_A s_2$  and both  $\text{LDTr}_A s_1$  and  $\text{LDTr}_A s_2$  are a closed term of an implicational type  $A_0$  whose order is less than four.

After obtaining two different closed terms  $\text{LDTr}_A s_1$  and  $\text{LDTr}_A s_2$  of the same implicational type  $A_0$  with order less than four using the proposition, we apply a term  $s'$  with poly-type  $A_0 \rightarrow B$ , which is defined in the next section, and we obtain

$$s'(\text{LDTr}_A s_1) =_{\beta\eta} t_1 \quad \text{and} \quad s'(\text{LDTr}_A s_2) =_{\beta\eta} t_2$$

such that two closed terms  $t_1$  and  $t_2$  of type  $B$  are outputs of the intended specification. This is an overview of our proof of Theorem 5.1 (Strong Typed Böhm Theorem). In order to construct the term  $s'$ , it is convenient to introduce a simple notion of model theory.

**Definition 3.2 (The Second-order Linear Term System)** (1) The language:

- (a) A denumerable set of variables  $\text{Var}$ : Elements of  $\text{Var}$  are denoted by  $x_1, x_2, \dots$
- (b) A denumerable set of second-order variables  $\text{SVar}$ : Elements of  $\text{SVar}$  are denoted by  $G_1, G_2, \dots$ . Each element of  $G$  of  $\text{SVar}$  has its arity  $\text{arity}(G) \geq 1$ .

(2) The set  $\text{SLT}$  of the terms of the language is defined inductively:

- (a) If  $x \in \text{Var}$  then  $x \in \text{SLT}$ .

- (b) If  $\{t_1, \dots, t_n\} \subseteq \text{SLT}$ ,  $G \in \text{SVar}$  has arity  $n$  and  $t_i$  and  $t_j$  have disjoint variables for each  $i, j$  ( $i \neq j$ ), then  $G(t_1, \dots, t_n) \in \text{SLT}$ .

(3) *Assignments:*

- (a) A variable assignment is a function  $\rho_1 : \text{Var} \rightarrow \{0, 1\}$ .
- (b) A second-order variable assignment is a function  $\rho_2$  from  $\text{SVar}$  to the set  $\text{CP}$ , where  $\text{CP}$  is the set of constant functions and (positive) projection functions on  $\{0, 1\}^n$  into  $\{0, 1\}$  for each  $n \geq 1$ .

(4) *Models:* A model for  $\text{SLT}$   $\llbracket - \rrbracket_{\langle \rho_1, \rho_2 \rangle} : \text{SLT} \rightarrow \{0, 1\}$  is determined uniquely for a given  $\langle \rho_1, \rho_2 \rangle$  as follows:

- (a)  $\llbracket x \rrbracket_{\langle \rho_1, \rho_2 \rangle} = \rho_1(x)$ .
- (b)  $\llbracket G(t_1, \dots, t_n) \rrbracket_{\langle \rho_1, \rho_2 \rangle} = \rho_2(G)(\llbracket t_1 \rrbracket_{\langle \rho_1, \rho_2 \rangle}, \dots, \llbracket t_n \rrbracket_{\langle \rho_1, \rho_2 \rangle})$ .

We note that in the definition above, to each second-order variable, a constant function or a (positive) projection is assigned. The following proposition is Proposition 25 in [13].

**Proposition 3.2** *Let  $s_1, s_2$  be in  $\text{SLT}$ . If  $s_1 \neq s_2$  then there are a variable assignment  $\rho_1$  and a second-order variable assignment  $\rho_2$  such that  $\llbracket s_1 \rrbracket_{\langle \rho_1, \rho_2 \rangle} \neq \llbracket s_2 \rrbracket_{\langle \rho_1, \rho_2 \rangle}$ .*

This proposition essentially uses linearity: for example we can not separate  $f(x)$  and  $f(f(f(x)))$  over  $\{0, 1\}$ . Then as observed in [13], we note that an implicational closed term  $s$  of a type  $A$  whose order is less than 4 is identified with an element  $s$  of  $\text{SLT}$ . So, without loss of generality, we can write  $s$  as a closed linear term

$$\text{fn } x_1 \Rightarrow \dots \Rightarrow \text{fn } x_n \Rightarrow \text{fn } G_1 \Rightarrow \dots \Rightarrow \text{fn } G_m \Rightarrow s_0$$

where the principal type of  $s$  has the following form:

$$\begin{array}{c} \overbrace{a_{01} \rightarrow \dots \rightarrow a_{0n}}^n \rightarrow \\ \quad \left( \overbrace{a_{11} \rightarrow \dots \rightarrow a_{1k_1}}^{k_1} \rightarrow a_{10} \right) \rightarrow \dots \rightarrow \left( \overbrace{a_{m1} \rightarrow \dots \rightarrow a_{mk_m}}^{k_m} \rightarrow a_{m0} \right) \\ \quad \rightarrow a_{00} . \end{array}$$

and each positive (resp. negative) occurrence of  $a_{ij}$  in the type has the corresponding exactly one negative (resp. positive) occurrence of  $a_{ij}$ . Unlike the weak typed Böhm theorem in [13], each  $a_{ij}$  will not be instantiated with the same type in main theorems in this paper: it may be instantiated with an implicational type with higher order. For this reason we need the notion of *poly-types*, which will be introduced in the next section.

## 4 Poly-Types

In this section we introduce the notion of poly-types, which is the key concept in this paper. For that purpose we need to introduce some notions.

**Principal Type Theorem** A *type substitution* is a function from type variables to types. It is well-known that any type substitution is uniquely extended to a function from types to types. A type  $A$  is an instance of a type  $B$  if there is a type substitution  $\theta$  such that  $A = B\theta$ . A type  $A$  is a principal type of a linear term  $t$  if (i) for some typing context  $\Gamma$ ,  $\Gamma \vdash t : A$  is derivable and (ii) when  $\Gamma' \vdash t : A'$  is derivable,

$A'$  and  $\Gamma'$  are an instance of  $A$  and  $\Gamma$  respectively. By the definition, if both  $A$  and  $A'$  are principal types of  $t$ , then  $A$  is an instance of  $A'$  and vice versa. So we can call  $A$  *the principal type* of  $t$  without ambiguity and write it as  $PT(t)$ . An untyped  $\lambda$ -term  $t$  is defined by the following syntax:

$$t ::= x \mid ts \mid \text{fn } x \Rightarrow t \mid (t, s) \mid \text{let val } (x, y) = s \text{ in } t$$

An untyped linear  $\lambda$ -term  $t$  is an untyped  $\lambda$ -term such that each free or bound variable in  $t$  occurs exactly once in  $t$ .

**Proposition 4.1** *If an untyped linear  $\lambda$ -term  $t$  is typable by the type assignment system in the previous section, then it has the principal type  $PT(t)$*

**Proof:** By assumption, we have a derivation for the term  $t$  with a type. Then by applying an easily modified version of the main result of [5] (see Section 7 of [5]) augmented with the  $*$  connective to  $t$ , we have a derivation for the term  $t$  with the principal type.  $\square$

Since our linear  $\lambda$ -calculus has the `let`-constructor and the  $(-, -)$  constructor, any untyped  $\lambda$ -term is not necessarily typable. A counterexample is `let val (x,y)=fn z=>z in (x, y)`. If the system has neither the `let`-constructor nor the  $(-, -)$  constructor, then any untyped  $\lambda$ -term is typable (see Theorem 4.1 of [8]).

## Poly-types

**Example 4.1** The following two terms are the basic constructs in [12]:

```
- fun True x y z = z x y;
- fun False x y z = z y x;
```

The terms `True` and `False` can be considered as the two normal terms of

$$\mathbb{B}_{\text{HM}} = 'a \rightarrow 'a \rightarrow ('a \rightarrow 'a \rightarrow 'a) \rightarrow 'a.$$

The following term can be considered as a *not* gate for  $\mathbb{B}_{\text{HM}}$ :

```
- fun Not_POLY p = p False True (fn f=>fn g=>(erase_3 g) f);
where
- fun I x = x;
- fun erase_3 p = p I I I;
```

We explain the reason in the following. The term `Not_POLY` has types  $A0 \rightarrow \mathbb{B}_{\text{HM}}$  and  $A1 \rightarrow \mathbb{B}_{\text{HM}}$ , where

$$\begin{aligned} A0 &= X0 \rightarrow Y0 \rightarrow (X0 \rightarrow Y0 \rightarrow Z0) \rightarrow Z0 & X0 &= A = Z0 \\ Y0 &= P \rightarrow (A \rightarrow A) \rightarrow (P \rightarrow P) \rightarrow (A \rightarrow A) \\ P &= (A \rightarrow A) \rightarrow (A \rightarrow A) & A &= \mathbb{B}_{\text{HM}} \\ A1 &= X1 \rightarrow Y1 \rightarrow (Y1 \rightarrow X1 \rightarrow Z1) \rightarrow Z1 & Y1 &= A = Z1 \\ X1 &= (A \rightarrow A) \rightarrow P \rightarrow (P \rightarrow P) \rightarrow (A \rightarrow A) \end{aligned}$$

Observe that  $A0 \neq A1$ . Moreover it is easy to see that there is no type substitution  $\theta$  such that  $\theta(A0) = \theta(A1)$ . On the other hand, two terms `True` and `False` have the principal types

$$'a \rightarrow 'b \rightarrow ('a \rightarrow 'b \rightarrow 'c) \rightarrow 'c \quad \text{and} \quad 'a \rightarrow 'b \rightarrow ('b \rightarrow 'a \rightarrow 'c) \rightarrow 'c,$$

respectively. Moreover, these types have instances  $A0$  and  $A1$  respectively. As a result, two application terms `Not_POLY True` and `Not_POLY False` have a type  $\mathbb{B}_{\text{HM}}$ .

Example 4.1 motivates the following definition.

**Definition 4.1** *Let  $t$  and  $s$  be two closed linear  $\lambda$ -terms such that  $\vdash t : A' \rightarrow B'$  and  $\vdash s : A$  are derivable and for some type substitution  $\theta_0$ ,  $\theta_0(\text{PT}(t)) = A_0 \rightarrow B$  and  $\theta_0(\text{PT}(s)) = A_0$ . Then we say that the term  $t$  is poly-typable by  $A \rightarrow B$  w.r.t.  $s$ .*

When  $t$  is poly-typable by  $A \rightarrow B$  w.r.t.  $s$ , observe that  $\vdash t : A \rightarrow B$  is not necessarily derivable. For example, the term `Not_POLY` is not typable by  $\mathbb{B}_{\text{HM}} \rightarrow \mathbb{B}_{\text{HM}}$ , but is poly-typable by  $\mathbb{B}_{\text{HM}} \rightarrow \mathbb{B}_{\text{HM}}$  w.r.t. `True` and `False` respectively. But then note that  $t s$  has type  $B$  in the usual sense. For example, both `Not_POLY True` and `Not_POLY False` have type  $\mathbb{B}_{\text{HM}}$ .

The importance of Definition 4.1 is the composability of two poly-typable terms. The proof of the following proposition is easy.

**Proposition 4.2** *Let  $t$  be poly-typable by  $A \rightarrow B$  w.r.t. two terms  $s$  and  $s'$  with type  $A$ . Moreover let  $t'$  be poly-typable by  $B \rightarrow C$  w.r.t. the two terms  $t s$  and  $t s'$ . Then the term  $\text{fn } x \Rightarrow (t' (t x))$  are poly-typable by  $A \rightarrow C$  w.r.t  $s$  and  $s'$ .*

We need a generalization of the definition above. Let  $t$  and  $s_i$  ( $1 \leq i \leq n$ ) be closed linear  $\lambda$ -terms such that  $\vdash t : A'_1 \rightarrow \dots \rightarrow A'_n \rightarrow B'$  and  $\vdash s_i : A_i$  are derivable. If for some type substitution  $\theta$ , we have  $\theta(\text{PT}(t)) = A'_1 \rightarrow \dots \rightarrow A'_n \rightarrow B$  and  $\theta(\text{PT}(s_i)) = A'_i$ , then we say that the term  $t$  is poly-typable by  $t : A_1 \rightarrow \dots \rightarrow A_n \rightarrow B$  w.r.t.  $s_i$ .

**Remark 1** *Poly-types are used in [12] without referring to it explicitly. Let  $A$  be a uniform data type consisting of exactly one type variable  $'a$  (for example,  $\mathbb{B}_{\text{HM}} = 'a \rightarrow 'a \rightarrow ('a \rightarrow 'a \rightarrow 'a) \rightarrow 'a$ ). In general, the principal type of a closed term of  $A$  is more general than  $A$ . The basic idea is to utilize the difference ingeniously. By using more general types, we can acquire more expressive power.*

## 5 Strong Typed Böhm Theorem

In this section we prove the first main theorem of this paper: a version of *the typed Böhm theorem* with regard to  $=_{\beta\eta c}$ . First we give some preliminary results, which state that for any types  $A$  and  $B$  having at least one closed term, we can always represent any projection from  $A \times \dots \times A$  to  $A$  and any constant function from  $A$  to  $B$  using the notion of poly-types.

**Lemma 5.1 (Projection Lemma)** *Let  $A$  be a type having at least one closed term. For any type  $B$ , there is a closed term  $t$  that is poly-typable by  $A \rightarrow (B \rightarrow B)$  w.r.t. any closed term  $s$  of  $A$  such that*

$$t s =_{\beta\eta c} I$$

**Proof:** The term  $t$  that we are looking for has the following form:

$$\text{fun } t \ x_0 = \text{LDTr\_A } x_0 \ \overbrace{I \ \dots \ I}^n \ \overbrace{u_1 \ \dots \ u_m}^m;$$

where  $\text{LDTr\_A}$  is the closed term obtained using Proposition 3.1 and the closed term  $u_j$  is defined by

$$\text{fun } u_j \ x_1 \dots x_{kj-1} \ x_{kj} = x_1 \ (\dots (x_{kj-1} \ (x_{kj} \ I)) \dots);$$

for each  $j$  ( $1 \leq j \leq m$ ). We note that the only occurrence of  $I$  in  $u_j$  is typed by  $'a \rightarrow 'a$  in the principal typing, which implies that it can be typed by  $B \rightarrow B$ . We also observe that the principal type of



LDTr\_A s0 has the following form:

$$\begin{aligned} & \overbrace{a_{01} \rightarrow \dots \rightarrow a_{0n}}^n \rightarrow \\ & (\overbrace{a_{11} \rightarrow \dots \rightarrow a_{1k_1}}^{k_1} \rightarrow a_{10}) \rightarrow \dots \rightarrow (\overbrace{a_{m1} \rightarrow \dots \rightarrow a_{mk_m}}^{k_m} \rightarrow a_{m0}) \\ & \rightarrow a_{00} . \end{aligned}$$

where each positive (resp. negative) occurrence of  $a_{ij}$  in the type has the corresponding exactly one negative (resp. positive) occurrence of  $a_{ij}$ . Since the combinator  $I$  is substituted for each bounded variables  $x_i$  ( $1 \leq i \leq k_j$ ) in  $u_j$ , the application term  $(ts)$  is reduced to  $I$ . Since the only occurrence of  $I$  in  $u_j$  can be typed by  $B \rightarrow B$ , the term  $(ts)$  can be typed by  $B \rightarrow B$ . This means that  $t$  can be poly-typed by  $A \rightarrow (B \rightarrow B)$  w.r.t. any closed term of type  $A$ .  $\square$

Note that a type variable  $a_{ij}$  may be instantiated with an implicational type of very higher order in the term  $t$ . For this reason we need the notion of *poly-types*.

The following corollary, which is a generalization of the proposition above to  $n$ -ary case, is obtained as a direct consequence of it.

**Corollary 5.1** *Let  $A$  be a type having at least one closed term. There is an  $i$ -th projection that is poly-typable by  $\overbrace{A \rightarrow \dots \rightarrow A}^n \rightarrow A$  for each  $i$  ( $1 \leq i \leq n$ ) and for any  $n$ .*

**Proof:** Think  $\overbrace{A \rightarrow \dots \rightarrow A}^n \rightarrow A$  as  $\overbrace{A \rightarrow \dots \rightarrow A}^{n-1} \rightarrow (A \rightarrow A)$ . Then let  $bx_i$  be

$$\text{LDTr\_A } x_i \overbrace{I \dots I}^{n_i} \overbrace{u_1 \dots u_{m_i}}^{m_i} ;$$

for  $i$  ( $0 \leq i \leq n-1$ ). The term  $t$  that we are looking for has the following form:

$$\text{fun } t \ x_0 \dots x_{n-1} \ x_n = bx_0 \ ( \dots (bx_{n-2} (bx_{n-1} \ x_n)) \dots );$$

$\square$

**Lemma 5.2 (Constant Function Lemma)** *Let  $A$  and  $B$  be types having at least one closed term. Let  $u$  be a closed term of  $B$ . Then there is a closed term  $t$  that is poly-typable by  $A \rightarrow B$  w.r.t. any closed term  $s$  of  $A$  such that*

$$ts =_{\beta\eta c} u$$

**Proof:** Let  $\text{proj}$  be the term which is poly-typable by  $A \rightarrow (B \rightarrow B)$  w.r.t. any closed term  $s$  of  $A$  obtained using Lemma 5.1. The term  $t$  that we are looking for is the following term:

$$\text{fun } t \ x_0 = \text{proj } x_0 \ u$$

$\square$

**Corollary 5.2** *Let  $A$  be a type having at least one closed term. Let  $s$  be such a closed term. There is a constant function that always returns  $s$  and is poly-typable by  $\overbrace{A \rightarrow \dots \rightarrow A}^n \rightarrow A$  for any  $n$ .*

**Theorem 5.1 (Strong Typed Böhm Theorem)** *For any types A and B, when any two different closed terms  $s_1$  and  $s_2$  of type A and any closed terms  $u_1$  and  $u_2$  of type B are given, there is a closed term  $t$  that is poly-typable by  $A \rightarrow B$  such that*

$$t \ s_1 =_{\beta_{\eta c}} u_1 \text{ and } t \ s_2 =_{\beta_{\eta c}} u_2$$

**Proof:** The term  $t$  that we are looking for has the following form:

$$\text{fun } t \ x_0 = \text{LDTr\_A } x_0 \ \overbrace{v_1 \cdots v_n}^n \ \overbrace{w_1 \cdots w_m}^m ;$$

By Proposition 3.1, we have  $\text{LDTr\_A } s_1 \neq_{\beta_{\eta c}} \text{LDTr\_A } s_2$ . Then since  $\text{LDTr\_A } s_1$  and  $\text{LDTr\_A } s_2$  are typable by a common type with order less than four, as observed before, they are identified with terms  $s_1$  and  $s_2$  in SLT respectively such that  $s_1 \neq s_2$ . Then by Proposition 3.2, there are a variable assignment  $\rho_1$  and a second-order variable assignment  $\rho_2$  such that  $\llbracket s_1 \rrbracket_{\langle \rho_1, \rho_2 \rangle} \neq \llbracket s_2 \rrbracket_{\langle \rho_1, \rho_2 \rangle}$ . Then following  $\rho_1$ , we choose  $u_1$  or  $u_2$  as the subterm  $v_i$  (with type B) of  $t$  for each  $i$  ( $1 \leq i \leq n$ ) and following  $\rho_2$ , we choose

a constant function or a projection as the subterm  $w_j$  (with poly-type  $\overbrace{B \rightarrow \cdots \rightarrow B}^{k_j} \rightarrow B$ ) of  $t$  for each  $j$  ( $1 \leq j \leq m$ ). These constant functions and projections are obtained using Projection and Constant Function Lemmas. Note that these constant functions and projections can be composed by Proposition 4.2 such that the closed term  $t$  is poly-typable appropriately. It is obvious that the term  $t$  has the desired properties.  $\square$

**Remark 2** *Theorem 5.1 can be considered as a strong version of Corollary 6 in [13]. While Corollary 6 in [13] uses only uniform type instantiation, Theorem 5.1 uses poly-types. We can not prove Theorem 5.1 using only uniform type instantiation. Appendix B gives a discussion of this matter.*

**Corollary 5.3** *Let  $s_1$  and  $s_2$  be two closed terms of A. Then there is a closed term  $\text{Copy\_A\_n}$  such that*

$$\text{Copy\_A\_n } s_1 =_{\beta_{\eta c}} (s_1, \dots, s_1) \quad \text{Copy\_A\_n } s_2 =_{\beta_{\eta c}} (s_2, \dots, s_2)$$

where  $s_1$  and  $s_2$  occur in  $(s_1, \dots, s_1)$  and  $(s_2, \dots, s_2)$   $n$  times respectively.

**Proof:** In Theorem 5.1, one chooses  $\overbrace{A * \cdots * A}^n$  as B, and then  $(s_1, \dots, s_1)$  and  $(s_2, \dots, s_2)$  as  $u_1$  and  $u_2$  respectively.  $\square$

The next theorem claims that in a limited situation we can obtain a closed term representing a function from closed terms of a type to closed terms that may not be typable by the same implicational type, but are *poly-typable* by the type.

**Theorem 5.2 (Poly-type Version of Strong Typed Böhm Theorem)** *Let  $s_1$  and  $s_2$  denote two different closed terms with type A, and  $u_1$  and  $u_2$  denote two different closed terms which are poly-typable by  $A_0 \rightarrow B$  w.r.t. two closed terms  $r_1$  and  $r_2$  with type  $A_0$  such that  $\{u_1 \ r_1, u_1 \ r_2, u_2 \ r_1, u_2 \ r_2\}$  is a set of one or two closed terms (with type B). Then there is a closed term  $t$  that is poly-typable by  $A \rightarrow A_0 \rightarrow B$  such that*

$$t \ s_1 \ r_i =_{\beta_{\eta c}} u_1 \ r_i \text{ and } t \ s_2 \ r_i =_{\beta_{\eta c}} u_2 \ r_i$$

for each  $i \in \{1, 2\}$ .

**Proof:** By Proposition 3.1 there is a linear  $\lambda$ -term  $\text{LDTr\_A}$  such that  $\text{LDTr\_A } s_1 \not\equiv_{\beta\eta c} \text{LDTr\_A } s_2$  and these terms can be regarded as different linearly labeled trees  $T_1$  and  $T_2$  respectively. In the rest of the proof, we assign a poly-typable first-order function to each leaf (which represented a first order variable in our proof of Theorem 5.1) and a poly-typable first-order or second-order function to each internal node (which represented a second order variable in our proof of Theorem 5.1) in  $T_1$  and  $T_2$ , following the structure of trees  $T_1$  and  $T_2$ . The purpose is to construct a closed term  $t$  such that each of  $t \leq_1$  and  $t \leq_2$  represents a one argument boolean function satisfying the specification of the theorem. The main tools are Projection and Constant Function Lemmas and the Strong Typed Böhm Theorem. We have two cases according to the structure of  $T_1$  and  $T_2$ .

- The case where both  $T_1$  and  $T_2$  have an  $n$ -ary second order variable  $F$  ( $n \geq 2$ ) and a first or second order variable  $G$  such that  $G$  is above  $F$  in both  $T_1$  and  $T_2$  and the position of  $G$  in  $T_1$  is different from that of  $T_2$ :

Furthermore, the case is divided into three cases. We assume that we choose  $F$  to be the nearest one to  $G$  in  $T_1$  and the variable in  $T_2$  that has the same position as  $G$  in  $T_1$  is  $H$ .

- The case where there is a path from the root to a leaf, including  $G$  in  $T_1$  such that the path does not include  $H$ , and when we interchange  $G$  and  $T_1$  with  $H$  and  $T_2$  respectively, the same thing happens:

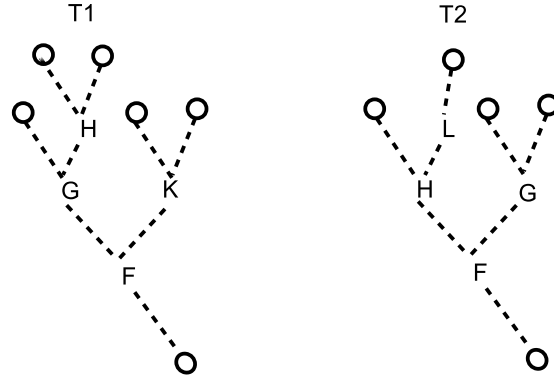
Without loss of generality, this case can be shown as Figure 1. The term  $t$  that we are looking for has the following form:

$$\begin{aligned} \text{fun } t \text{ } x_0 \text{ } y_0 = \\ \text{let val } (x_1, \dots, x_n) = \text{Copy\_A0\_n } y_0 \text{ in} \\ \text{LDTr\_A } x_0 \overbrace{(v_1 \ x_1) \dots (v_n \ x_n)}^n \overbrace{w_1 \dots w_m}^m \text{ end;} \end{aligned}$$

where the subterm  $v_i$  that is poly-typable by  $\text{A0} \rightarrow B$  is obtained using Theorem 5.1, representing a surjection from  $\{r_1, r_2\}$  to one or two element set  $\{u_1 \ r_1, u_1 \ r_2, u_2 \ r_1, u_2 \ r_2\}$  for each  $i$  ( $1 \leq i \leq n$ ). The subterm  $w_j$  that is poly-typable by

$\overbrace{B \rightarrow \dots \rightarrow B}^{k_j} \rightarrow B$  for each  $j$  ( $1 \leq j \leq m$ ) is constructed from Projection Lemma w.r.t. an appropriate position except for  $G$  and  $H$ . For example the first argument projection is assigned to  $F$  in Figure 1. Then  $G$  and  $H$  are constructed in the following two steps:

1. First we construct terms  $m_j$  with type  $B \rightarrow B$  using the Strong Typed Böhm Theorem (Theorem 5.1). The functions for  $G$  and  $H$  are the constant, identity, or negation functions, depending on  $u_1$  and  $u_2$ . Note that in order to represent the negation function we need the Strong Typed Böhm Theorem.
  2. Second from using  $m_j$ , we construct  $w_j$  using Constant Function Lemma in order to discard the unnecessary arguments. The terms corresponding to  $G$  and  $H$  in Figure 1 discard the second argument.
- The case where (i) there is no any path from the root to a leaf, including  $G$  in  $T_1$  such that the path does not include  $H$  and (ii) there is a path from the root to a leaf, including  $H$  in  $T_2$  such that the path does not include  $G$ :  
We assume that the variable in  $T_1$  that has the same position as  $G$  in  $T_2$  is  $K$ . In this case, the following additional properties hold:  
(iii) There is no any path from the root to a leaf, including  $G$  in  $T_2$  such that the path does not include  $K$ .



There is a path from the root to a leaf, including  $G$  in  $T_1$  such that the path does not include  $H$ , and when we interchange  $G$  and  $T_1$  with  $H$  and  $T_2$  respectively, the same thing happens.

Figure 1: Two different linearly labeled trees (1)

- (iv) there is a path from the root to a leaf, including  $K$  in  $T_1$  such that the path does not include  $G$ .

Otherwise, we can apply the immediately above case (replace  $G$  and  $H$  by  $K$  and  $G$  respectively). In the case,  $T_1$  and  $T_2$  have the form of Figure 2 or Figure 3 without loss of generality. First we consider the case of Figure 2. The term  $t$  that we are looking for has the following form:

```

fun t x0 y0 =
  let val (x1, ..., xn) = Copy_A0_n y0 in
    LDTr_A x0  $\overbrace{(v1 \ x1) \cdots (vn \ xn)}^n$   $\overbrace{w1 \cdots wm}^m$  end;

```

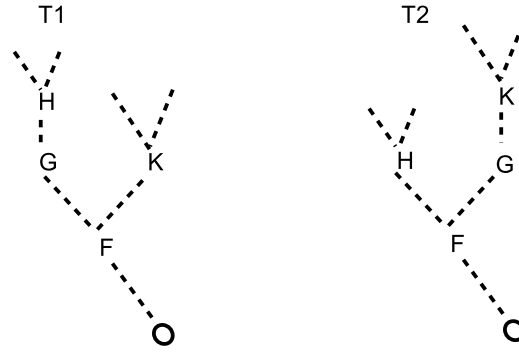
where the subterm  $v_i$  that is poly-typable by  $A_0 \rightarrow B$  is obtained using the Strong Typed Böhm Theorem (Theorem 5.1) for each  $i$  ( $1 \leq i \leq n$ ), representing a surjection from  $\{r_1, r_2\}$  to one or two element set  $\{u_1 r_1, u_1 r_2, u_2 r_1, u_2 r_2\}$  and the subterm  $w_j$  is

poly-typable by  $\overbrace{B \rightarrow \cdots \rightarrow B}^{k_j} \rightarrow B$  for each  $j$  ( $1 \leq j \leq m$ ) obtained from Projection Lemma except that four terms assigned to  $F$ ,  $G$ ,  $H$ , and  $K$  are selected according to the table immediately below (and then Constant Function Lemma is applied in order to discard the

unnecessary arguments):

$u_1$	$u_2$	argument choice of $F$	$G$	$H$	$K$
const.	const.	left	const.	const.	don't care
const.	id.	left	const.	id.	don't care
const.	neg.	left	const.	neg.	don't care
id.	const.	right	const.	don't care	id.
neg.	const.	right	const.	don't care	neg.
id.	id.	left	id.	id.	id.
neg.	neg.	left	id.	neg.	don't care
id.	neg.	left	neg.	neg.	don't care
neg.	id.	left	neg.	id.	don't care

where id., neg., and const. mean the identity, negation, and constant functions respectively. The term “don't care” means that we can choose any one argument function for that place. In the case of Figure 3, the form of the term  $t$  is the same as Figure 2. The only difference is that we assign one argument functions to the subterms  $v$  is corresponding to  $x$  and  $y$ , according to the instructions for  $H$  and  $K$  in the above table respectively. We can do the assignment using Theorem 5.1.



(i) There is no any path from the root to a leaf, including  $G$  in  $T_1$  such that the path does not include  $H$  and (ii) there is a path from the root to a leaf, including  $H$  in  $T_2$  such that the path does not include  $G$ .

Figure 2: Two different linearly labeled trees (2)

– Otherwise:

In this case, any path from the root to a leaf including  $G$  (resp.  $H$ ) in  $T_1$  (resp.  $T_2$ ) includes  $H$  (resp.  $G$ ) above  $G$  (resp.  $H$ ). Without loss of generality, this case can be shown as Figure 4. The term  $t$  that we are looking for has the following form:

$$\text{fun } t \text{ } x_0 \text{ } y_0 = \text{LDTr\_A } x_0 \text{ } \overbrace{v_1 \cdots v_n}^n \text{ } \overbrace{w_1 \cdots w_m}^m \text{ } (t_0 \text{ } y_0);$$

where  $t_0$  that is poly-typable by  $A_0 \rightarrow B$  is obtained from the Strong Typed Böhm Theorem (Theorem 5.1) which represents a surjection from  $\{r_1, r_2\}$  to one or two element set

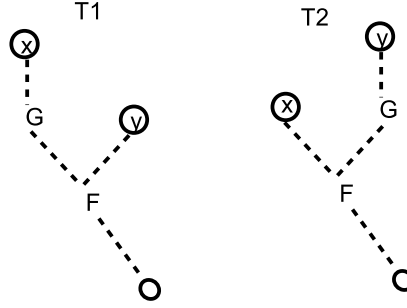


Figure 3: Two different linearly labeled trees (3)

$\{u_1 r_1, u_1 r_2, u_2 r_1, u_2 r_2\}$ , the subterm  $v_i$  is poly-typable by  $B \rightarrow B$  obtained from Constant Function Lemma for each  $i$  ( $1 \leq i \leq n$ ), and the subterm  $w_j$  has type

$\overbrace{C_1 \rightarrow \dots \rightarrow C_{k_j}}^{k_j} \rightarrow D$  for each  $j$  ( $1 \leq j \leq m$ ) where  $C_i$  and  $D$  is poly-typable by  $B \rightarrow B$ . The subterm  $w_j$  is constructed from Projection Lemma w.r.t. an appropriate position except for  $G$  and  $H$ . For example, in Figure 4, the first projection function is assigned to  $F$ . The terms  $G$  and  $H$  are constructed by the following two steps:

1. First we construct a term  $m_j$  with type  $D$  using the Strong Typed Böhm Theorem (Theorem 5.1). The functions for  $G$  and  $H$  are the constant, identity, or negation functions, depending on  $u_1$  and  $u_2$ . Note that in order to represent the negation function we need the Strong Typed Böhm Theorem.
2. Second from using  $m_j$ , we construct  $w_j$  using Constant Function Lemma in order to discard the unnecessary arguments.

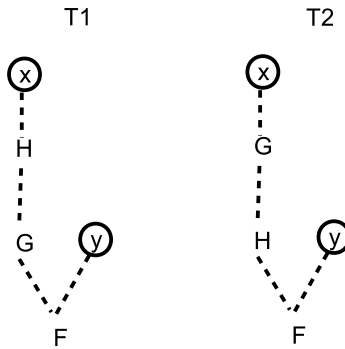


Figure 4: Two different linearly labeled trees (4)

- Otherwise:

The case is any of the degenerated versions of the cases above. We can apply the same discussion.

□

## 6 Functional Completeness of Linear Types: An Application of Strong Typed Böhm Theorem

Strong typed Böhm theorem for the linear  $\lambda$ -calculus is not a theoretical non-sense. It has an algorithmic content and at least one application: functional completeness of linear types.

**Definition 6.1** *Let  $A$  be a type that has two different closed terms  $s1$  and  $s2$ . A function  $f : \{0,1\}^n \rightarrow \{0,1\}$  is represented by a closed term  $t$  that is poly-typable by  $\overbrace{A \multimap \dots \multimap A}^n \multimap A$  with regard to  $s1$  and  $s2$  if, for any  $\langle x_1, \dots, x_n \rangle \in \{0,1\}^n$  and  $y \in \{0,1\}$*

$$f(x_1, \dots, x_n) = y \Leftrightarrow t \ x1 \cdots xn =_{\beta\eta c} y$$

where  $x1, \dots, xn, y$  are the images of  $x_1, \dots, x_n, y$  under the map  $\{0 \mapsto s1, 1 \mapsto s2\}$  respectively. The type  $A$  is functionally complete with regard to  $s1$  and  $s2$  if any function  $f : \{0,1\}^n \rightarrow \{0,1\}$  is represented by a closed term with regard to  $s1$  and  $s2$ .

The following proposition is well-known.

**Proposition 6.1** *A type  $A$  is functionally complete if and only if the Boolean not gate, the and gate, and the duplicate function, i.e.,  $\{0 \mapsto \langle 0,0 \rangle, 1 \mapsto \langle 1,1 \rangle\}$  are represented over  $A$ .*

So far Mairson [12] gave the functional completeness of type  $\mathbb{B}_{HM} = 'a \multimap 'a \multimap ('a \multimap 'a \multimap 'a) \multimap 'a$  with regard to the two closed terms. Moreover van Horn and Mairson [9] gave the functional completeness of  $\mathbb{B}_{TWIST} * \mathbb{B}_{TWIST}$  with regard to its two closed terms, where  $\mathbb{B}_{TWIST} = 'a * 'a \multimap 'a * 'a$ . In fact, the following theorem holds.

**Theorem 6.1** *Let  $A$  be a type that has two different closed terms  $s1$  and  $s2$ . Then the type  $A$  is functionally complete with regard to  $s1$  and  $s2$ .*

**Proof:** The representability of the *not* gate and the duplicate function are a direct consequence of strong typed Böhm theorem: while in the *not* gate we choose  $A$  as  $B$  in Theorem 5.1 and  $s2$  and  $s1$  as  $u1$  and  $u2$  respectively, in the duplicate function we choose  $A * A$  as  $B$  and  $(s1, s1)$  and  $(s2, s2)$  as  $u1$  and  $u2$  respectively.

On the other hand, by Constant Function Lemma (Lemma 5.2), there is a term  $t$  with poly-type  $A \multimap A$  that represents the constant function  $\{0 \mapsto 0, 1 \mapsto 0\}$ . Then we choose  $A \multimap A$  as  $A0 \multimap B$  in Theorem 5.2 and we choose  $t$  and  $I = \text{fn } x \Rightarrow x$  as  $u1$  and  $u2$  respectively. Then we get a term  $t'$  that represents the *and* gate.  $\square$

Appendix C gives a functional completeness proof of  $\mathbb{B}_{HM}$ , which is extracted from proofs shown above and is slightly different from that of [12]. Note that our construction of functional completeness is not compatible with the polymorphic  $\lambda$ -calculus by Girard and Reynolds (for example, see [7, 4]): For example,  $\text{Not}_{HM}$  can not be typed by  $\forall 'a. \mathbb{B}_{HM} \multimap \forall 'a. \mathbb{B}_{HM}$ . As far as we know, the only type that is compatible with the polymorphic  $\lambda$ -calculus is  $\mathbb{B}_{Seq} = 'a \multimap ('a \multimap 'a) \multimap ('a \multimap 'a) \multimap 'a$ . Appendix D gives the functional completeness proof of  $\mathbb{B}_{Seq}$  that is compatible with the polymorphic lambda calculus. While the encoding derived from our proof of Theorem 6.1 is not compatible with the calculus, the modified version given in Appendix D is compatible. It would be interesting to pursue this topic, i.e., whether or not other types are compatible with the polymorphic  $\lambda$ -calculus.

## 7 Concluding Remarks

With regard to the functional completeness problem of the linear  $\lambda$ -calculus, Theorem 6.1 is not the end of the story. For example, we have already found some better Boolean encodings than that given by Theorem 6.1 (see Appendix D and [14]). We should discuss efficiency of various Boolean encodings in the linear  $\lambda$ -calculus and relationships among them. Moreover the extension to  $n$ -valued cases instead of the 2-valued Boolean case is open. Our result is the first step toward these research directions.

**Acknowledgments.** The author thanks an anonymous referee, who pointed out the simplified definition of the relation  $\leftrightarrow_c$ .

## References

- [1] H. Barendregt, W. Dekkers & R. Statman (2013): *Lambda Calculus with Types*. Cambridge University Press, doi:10.1017/CB09781139032636.
- [2] H. P. Barendregt (1981): *The Lambda Calculus: Its Syntax and Semantics*. North Holland.
- [3] R. Blute & P. Scott (2004): *Category Theory for Linear Logicians*, pp. 3–64. *LMS Lecture Note Series* 316, Cambridge University Press, doi:10.2277/0521608570.
- [4] R. L. Crole (1994): *Categories for Types*. Cambridge University Press, doi:10.1017/CB09781139172707.
- [5] L. Damas & R. Milner (1982): *Principal Type-Schemes for Functional Programs*. In: *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*, pp. 207–212, doi:10.1145/582153.582176.
- [6] J.-Y. Girard (1987): *Linear Logic*. *Theoretical Computer Science* 50, pp. 1–102, doi:10.1016/0304-3975(87)90045-4.
- [7] J.-Y. Girard, Y. Lafont & P. Taylor (1989): *Proofs and Types*. Cambridge University Press.
- [8] R. Hindley (1989): *BCK-combinators and Linear  $\lambda$ -terms have Types*. *Theoretical Computer Science* 64, pp. 97–105, doi:10.1016/0304-3975(89)90100-X.
- [9] D. Van Horn & H. G. Mairson (2007): *Relating complexity and precision in control flow analysis*. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pp. 85–96, doi:10.1145/1291151.1291166.
- [10] J. Lambek & P. Scott (1988): *Introduction to Higher-Order Categorical Logic*. Cambridge University Press.
- [11] I. Mackie, L. Román & S. Abramsky (1993): *An Internal Language for Autonomous Categories*. *Applied Categorical Structures* 1, pp. 311–343, doi:10.1007/BF00873993.
- [12] H. G. Mairson (2004): *Linear Lambda Calculus and PTIME-completeness*. *Journal of Functional Programming* 14(6), pp. 623–633, doi:10.1017/S0956796804005131.
- [13] S. Matsuoka (2007): *Weak Typed Böhm Theorem on IMLL*. *Annals of Pure and Applied Logic* 145(1), pp. 37–90, doi:10.1016/j.apal.2006.06.001.
- [14] S. Matsuoka (2015): *A New Proof of P-time Completeness*. In Geoff Sutcliffe Ansgar Fehnker, Annabelle McIver & Andrei Voronkov, editors: *LPAR-20. 20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations, EPIc Series in Computer Science* 35, EasyChair, pp. 119–130. Available at [http://www.easychair.org/publications/download/A\\_New\\_Proof\\_of\\_P-time\\_Completeness\\_of\\_Linear\\_Lambda\\_Calculus](http://www.easychair.org/publications/download/A_New_Proof_of_P-time_Completeness_of_Linear_Lambda_Calculus).
- [15] R. Milner, M. Tofte, R. Harper & D. MacQueen (1997): *The Definition of Standard ML (Revised)*. MIT Press.
- [16] R. Statman (1980): *On the existence of closed terms in the typed  $\lambda$ -calculus. I*, pp. 511–534. Academic Press.



- [17] R. Statman (1983):  *$\lambda$ -definable Functionals and  $\beta\eta$ -conversion*. *Archiv für mathematische Logik und Grundlagenforschung* 23, pp. 21–26, doi:10.1007/BF02023009.
- [18] R. Statman & G. Dowek (1992): *On Statman's Finite Completeness Theorem*. Technical Report, Carnegie Mellon University. CMU-CS-92-152.
- [19] A. S. Troelstra (1992): *Lectures on Linear Logic*. CSLI.

## A The relationship between linear $\lambda$ terms and IMLL proof nets

### A.1 Brief Introduction to IMLL proof nets

In this appendix, we introduce IMLL proof nets briefly. For a complete treatment, for instance see [13].

**Definition A.1 (Plain and signed IMLL formulas)** *The plain IMLL formulas are defined in the following grammar:*

$$A ::= p \mid A \otimes B \mid A \multimap B$$

where  $p$  is called a propositional variable. A signed IMLL formula has the form  $A^+$  or  $A^-$ , where  $A$  is a plain IMLL formula.

**Definition A.2 (Links)** *A link is an object with a few signed IMLL formulas. Any link is any of ID-,  $\otimes^+$ -,  $\otimes^-$ -,  $\multimap^+$ -, or  $\multimap^-$ -link shown in Figure 5.*

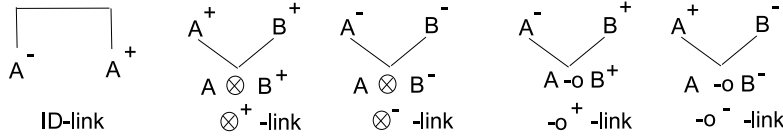


Figure 5: Links

**Definition A.3 (IMLL proof nets)** *An IMLL proof net is defined inductively as shown in Figure 6.*

**Definition A.4 (Reduction rules)** *Reduction rules for an IMLL proof net have two kinds: one is multiplicative shown in Figure 7 and the other  $\eta$  shown in Figure 8.*

The reduction relation over IMLL proof nets induced by these reduction rules is strong normalizing and confluent. So we can obtain a unique normal form of any IMLL proof net. For two IMLL proof nets  $\Theta_1$  and  $\Theta_2$ ,  $\Theta_1$  is equal to  $\Theta_2$  (denoted by  $\Theta_1 = \Theta_2$ ) if there is a bijective map from the signed IMLL formula occurrences in the normal form of  $\Theta_1$  to that of  $\Theta_2$  such that the map preserves the link structure (for the complete treatment, see [13]).

### A.2 A full and faithful embedding of the linear $\lambda$ -calculus into IMLL proof nets

First we define our translation  $\llbracket - \rrbracket$  of linear  $\lambda$ -terms into IMLL proof nets by Figure 9, where we identify IMLL proof nets up to  $=$  defined by Definition 14 in [13] (or Appendix A.1). Then the following proposition holds.

**Proposition A.1** *If  $t \rightarrow_{\beta\eta c} t'$  then,  $\llbracket t \rrbracket = \llbracket t' \rrbracket$ .*

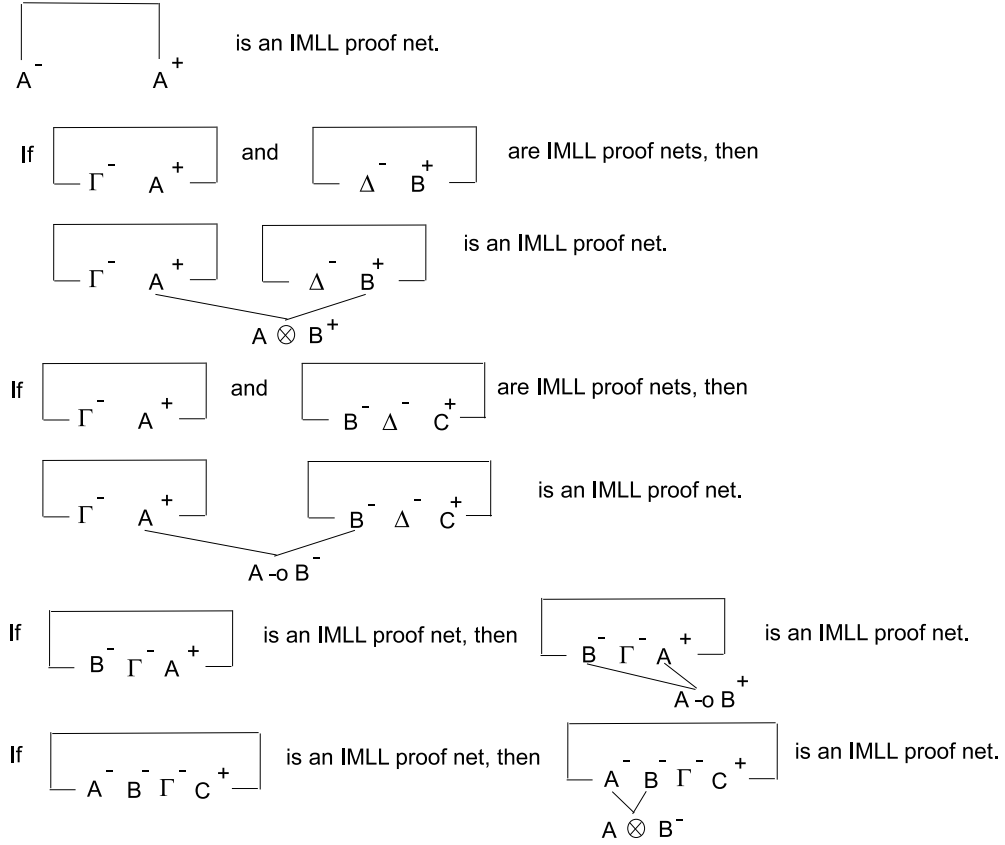


Figure 6: IMLL proof nets

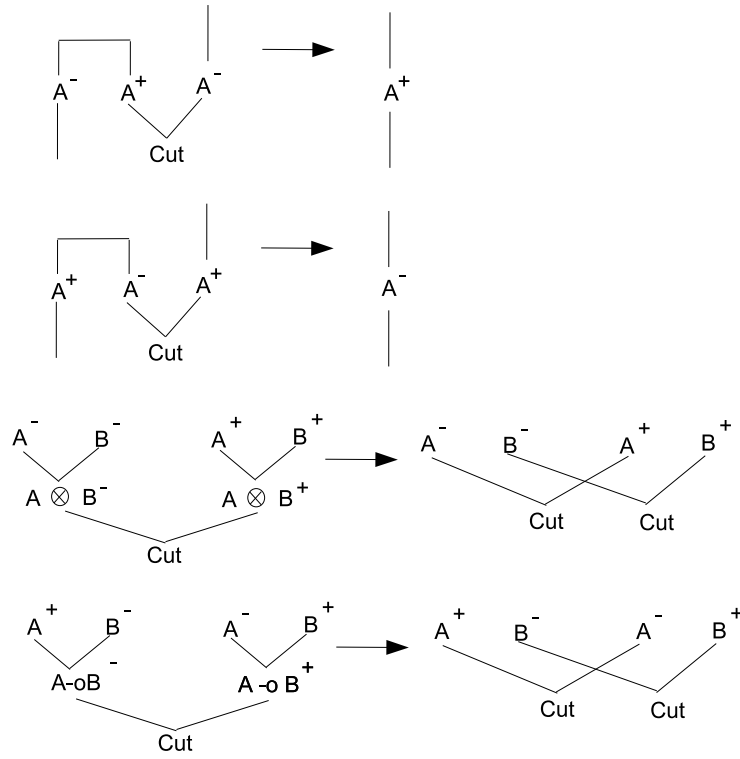
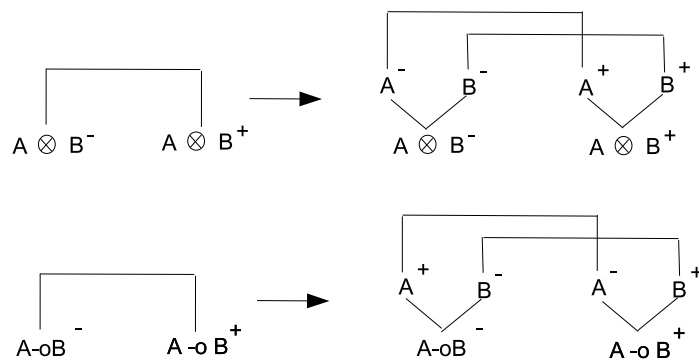
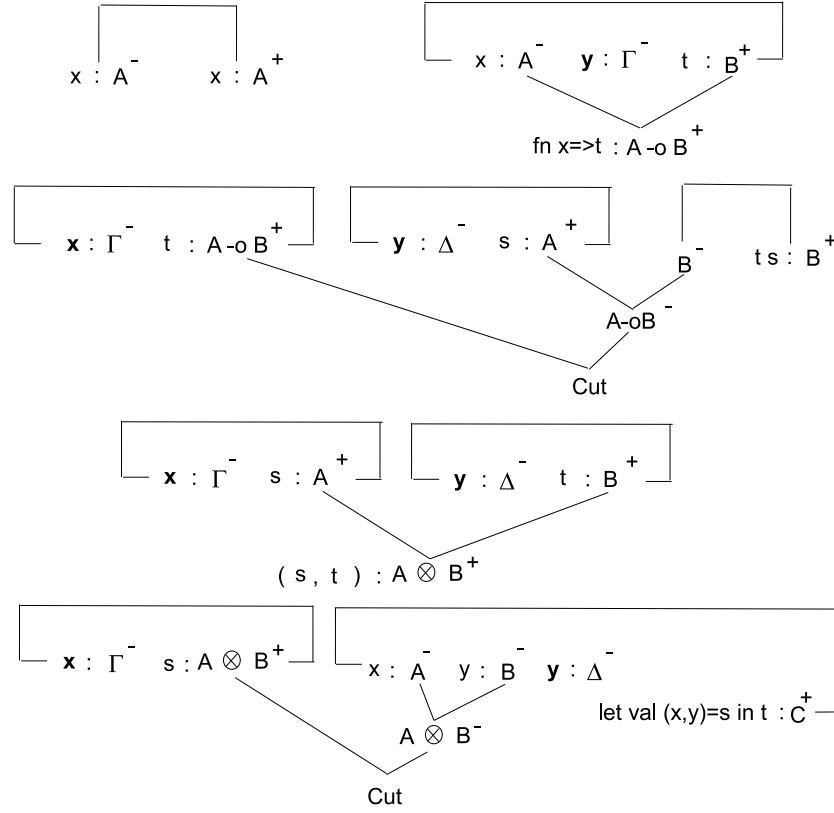
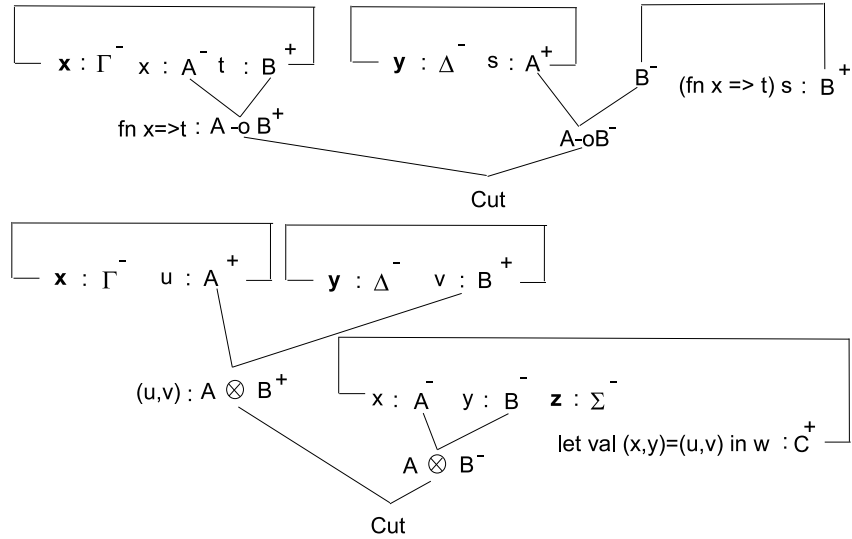
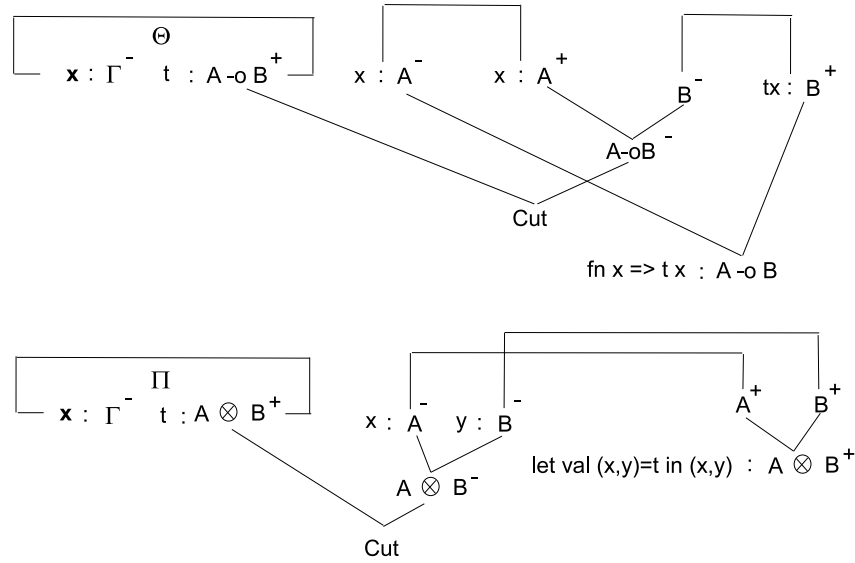


Figure 7: Multiplicative reduction rules

Figure 8:  $\eta$  reduction rules

Figure 9: Translation of Linear  $\lambda$ -Terms into IMLL proof netsFigure 10: Translation of  $\beta$ -redexes

Figure 11: Translation of  $\eta$ -redexes

**Proof:** When  $t \rightarrow_\beta t'$ , Figure 10 proves the proposition. When  $t \rightarrow_\eta t'$ , we consider Figure 11. In Figure 11, we normalize IMLL proof nets  $\Theta$  and  $\Pi$ . Then the proposition should be obvious. When  $t \leftrightarrow_c t'$ ,  $t$  and  $t'$  are translated into the same IMLL proof net in each case.  $\square$

Moreover if both  $t$  and  $t'$  are normal forms of linear  $\lambda$ -terms with regard to  $\rightarrow_{\beta\eta c}$ , then when  $\neg(t =_{\beta\eta c} t')$ , it is obvious that  $\llbracket t \rrbracket \neq \llbracket t' \rrbracket$ . So we have established the faithfulness. On the other hand, for any IMLL proof net  $\Theta$  whose conclusion is a type of the linear  $\lambda$ -calculus, it is easy to show that there is a linear  $\lambda$ -term  $t$  such that  $\llbracket t \rrbracket = \Theta$ . So we have established the fullness. Therefore we conclude the existence of a full and faithful embedding stated above. So we can identify a normal linear  $\lambda$ -term with the corresponding normal IMLL proof net. We treat  $=_{\beta\eta c}$  as the legitimate equality of linear  $\lambda$ -terms. Note that while  $\eta$ -normal forms are natural in the linear  $\lambda$ -calculus,  $\eta$ -long normal forms are natural in the proof net formalism.

## B Why Need Poly-Types?

In this appendix, we show that the method of [13] can not be extended without poly-types.

We let  $\mathbb{B}_{\text{HM}} = 'a \multimap 'a \multimap ('a \multimap 'a \multimap 'a) \multimap 'a$  and  $\mathbb{B}_{\text{Seq}} = 'a \multimap ('a \multimap 'a) \multimap ('a \multimap 'a) \multimap 'a$  and

```

fun True x y z = z x y;
fun False x y z = z y x;
fun TrSeq x f g = g (f x);
fun FlSeq x f g = f (g x);

```

The terms `True` and `False` are closed terms of  $\mathbb{B}_{\text{HM}}$  and `TrSeq` and `FlSeq` are that of  $\mathbb{B}_{\text{Seq}}$ . Then we show that for any type  $A$ , we cannot find a closed term  $s$  of type  $\mathbb{B}_{\text{Seq}}[A/'a] \multimap \mathbb{B}_{\text{HM}}$  such that

$$s \text{ TrSeq} =_{\beta\eta c} \text{True} \quad \text{and} \quad s \text{ FlSeq} =_{\beta\eta c} \text{False}.$$

We suppose that there is such a closed term  $s$ . Then  $A$  must be  $\mathbb{B}_{\text{HM}}$ . Moreover there must be closed

terms  $f$  and  $g$  of type  $\mathbb{B}_{\text{HM}} \rightarrow \mathbb{B}_{\text{HM}}$  such that

$$f(gt) =_{\beta\eta c} \text{True} \quad \text{and} \quad g(ft) =_{\beta\eta c} \text{False}$$

where  $t$  is  $\text{True}$  or  $\text{False}$ . But  $f$  and  $g$  must be *identity* or *not gate*, because  $\mathbb{B}_{\text{HM}} \rightarrow \mathbb{B}_{\text{HM}}$  does not allow any constant functions. This is impossible.

## C Functional Completeness of $\mathbb{B}_{\text{HM}}$

The terms  $\text{Not\_HM}$ ,  $\text{Copy\_HM}$ ,  $\text{And\_HM}$  below are derived from our construction.

```
fun True x y z = z x y;
fun False x y z = z y x;
fun I x = x;
fun u_2 x1 x2 = x1 (x2 I);
fun u_3 x1 x2 x3 = x1 (x2 (x3 I));
fun proj_1 x1 x2 = x2 I I u_2 x1;
fun Not_HM x = x False True proj_1;
fun LDTr_Pair p x y f z w h l
= let val (u,v) = p in l (u x y f) (v z w h) end;
fun proj_Pair_1 x1 x2 = LDTr_Pair x2 I I u_2 I I u_2 u_2 x1;
fun Copy_HM x = x (True,True) (False,False) proj_Pair_1;
fun const_F x = x I I (u_2) False;
fun And_HM x y = let val (u,v) = Copy_HM y in
x (I u) (const_F v) proj_1 end;
```

## D Functional Completeness of $\mathbb{B}_{\text{seq}}$

The terms  $\text{NotSeq}$ ,  $\text{CopySeq}$ ,  $\text{AndSeq}$  below are compatible with the polymorphic lambda calculus of Girard-Reynolds.

```
fun TrSeq x f g = g (f x);
fun FlSeq x f g = f (g x);
fun NotSeq h x f g = h x g f;
fun constTr h x f g = g (f (h x I I));
fun conv h z = let val (f,g) = h in let val (x,y) = z
in (f x, g y) end end;
fun CopySeq x =
x (TrSeq,TrSeq) (conv (NotSeq,NotSeq)) (conv (constTr,constTr));
fun constFlFun h k x f g = f (g (k (h FlSeq x I I) I I));
fun idFun h k x f g = k (h TrSeq x I I) f g;
fun AndSeq x = x I constFlFun idFun;
```